System V Application Binary Interface

x86-64^{TM} Architecture Processor Supplement

Draft Version 0.10

Edited by
Jan Hubicka[1], Andreas Jaeger[2], Mark Mitchell[3]

August 15, 2000

[1]jh@suse.cz
[2]aj@suse.de
[3]mark@codesourcery.com

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Except where otherwise noted, the x86-64 architecture follows the conventions described in the Intel386 psABI. Rather than replicate the entire contents of the Intel386 psABI document, this document simply indicates the differences from that document. The ABI described in this document applies only to the x86-64 architecture when it is executing in 32/64-bit mode; when the architecture executes in ordinary 32-bit mode the Intel386 psABI conventions apply.

As with all psABI documents, no attempt has been made to specify an ABI for languages other than C. However, it is assumed that many programming languages will wish to link with code written in C, so that the ABI specifications documented here are relevant.[1]

1.1 Differences from the Intel386 psABI

The most fundamental differences from the Intel386 psABI document are as follows:

- Changes to sizes of fundamental data types.
- Changes to calling conventions.
- Changes to the floating-point support.
- Removal of the GOT register.

[1]For C++, a variant of the ABI developed for use on the Intel IA64 architecture will likely be used.

Chapter 2

Software Installation

Not yet done.

Chapter 3

Low Level System Information

3.1 Machine Interface

3.1.1 Processor Architecture

The x86-64 architecture (see `http://www.x86-64.org/documentation`) defines extensions to the x86 architecture. The x86-64 architecture allows the execution of 32-bit programs in 32-bit mode. Such programs will continue to make use of the Intel386 psABI; this document applies only to programs running in "long" mode.

3.1.2 Data Representation

Within this specification, the term *halfword* refers to a 16-bit object, the term *word* refers to a 32-bit object, the term *doubleword* refers to a 64-bit object, and the term *quadword* refers to a 128-bit object.

Fundamental Types

Figure 3.1 shows the correspondence between ISO C's scalar types and the processor's.

The `__float128` type uses a 15-bit exponent, a 113-bit mantissa (the high order significant bit is implicit) and an exponent bias of 16383.[1]

[1] Initial implementations of the x86-64 architecture are expected to support operations on the `__float128` type only via software emulation.

Figure 3.1: Scalar Types

Type	C	sizeof	Alignment (bytes)	x86-64 Architecture
Integral	`char` `signed char`	1	1	signed byte
	`unsigned char`	1	1	unsigned byte
	`short` `signed short`	2	2	signed halfword
	`unsigned short`	2	2	unsigned halfword

	`int`
	`signed int`
	`enum`
		4	4	signed word

	`unsigned int`	4	4	unsigned word

	`long`
	`signed long`
	`long long`
	`signed long long`
		8	8	signed doubleword

	`unsigned long` `unsigned long long`	8 8	8 8	unsigned doubleword unsigned doubleword
Pointer	*any-type* `*` *any-type* `(*)()`	8	8	unsigned doubleword

Floating-point	`float`
	`double`
	`long double`
	`__float128`
		4
		8
		16
		16
			4
			8
			16
			16
				single (IEEE)
				double (IEEE)
				80-bit extended (IEEE)
				128-bit extended (IEEE)

Packed	`__m64` `__m128`	8 16	8 16	*MMX* and 3DNow! SSE and SSE-2

The `long double` type uses a 15 bit exponent, a 64-bit mantissa with an explicit high order significand bit and an exponent bias of 16383[2]. The contents of the 6 padding bytes is undefined.

A null pointer (for all types) has value zero.

Like the Intel386 architecture, the x86-64 architecture does not require all data access to be properly aligned. Accessing misaligned data will be slower than accessing properly aligned data, but otherwise there is no difference.

Aggregates And Unions

An array uses the same alignment as its elements, except that an array that requires at least 16 bytes always has alignment of at least 16 bytes.[3]

No other changes required.

Bit-Fields

Amend the description of bit-field ranges as follows:

Figure 3.2: Bit-Field Ranges

Bit-field Type	Width w	Range
`signed long`		-2^{w-1} to $2^{w-1}-1$
`long`	1 to 64	0 to 2^w-1
`unsigned long`		0 to 2^w-1

Properties of bitfields having the type `__m64` or `__m128` are not defined by the ABI. Although compilers are free to support bitfields with these types, programs making use of these features will not conform to the x86-64 ABI.

No other changes required.

3.2 Function Calling Sequence

This section describes the standard function calling sequence, including stack frame layout, register usage, parameter passing and so on.

[2] This is the x87 double extented precision data type.

[3] This alignment requirement allows the use of SSE operations.

The standard calling sequence requirements apply only to global functions. Local functions that are not reachable from other compilation units may use different conventions. Nevertheless, it is recommended that all functions use the standard calling sequence when possible.

3.2.1 Registers

The x86-64 architecture provides 16 general purpose 64-bit registers. In addition the architecture provides 16 SSE registers, each 128 bits wide and 8 x87 floating point registers, each 80 bits wide. Each of the x87 floating point registers may be referred to in *MMX*/3DNow! mode as a 64-bit register. There are several additional special-purpose registers.

This subsection discusses usage of each register. The registers called *global registers* "belong" to the calling function and the called function is required to preserve their values, while the *temporary registers* may be freely clobbered by the called function. (*Editor:* The split to global/temporaries is still subject of change.)

General Purpose Registers

RAX Temporary register used to pass first integer argument and to return the first doubleword of the return value.

RDX Temporary register used to pass second integer argument and to return the second doubleword of the return value.

RCX Temporary register used to pass third integer argument.

RBX Temporary register used to pass fourth integer argument.

RBP Global register optionally used as frame pointer.

RSP The stack pointer.

RSI Temporary register used to pass fifth doubleword of argument area.

RDI Temporary register used to pass sixth doubleword of argument area.

R8, R9 Temporary registers.

R10 – R15 Global registers.

SSE registers

`XMM0`, `XMM1` Temporary registers used to pass first two SSE arguments and to return first two doublewords of the return value.

`XMM2` – `XMM6` Temporary registers used to pass 2nd–6th SSE arguments

`XMM6` – `XMM13` Global registers

`XMM13` – `XMM15` Temporary registers

MMX/3DNow! and x87 registers

Since the usage of SSE instruction set if prefered over the x87, the CPU is in "*MMX*" mode across function calls. Every function that uses x87 register stack is required to convert the registers properly using `emms` or `femms` instructions and thus x87 registers cannot be used to hold values across a function call. To make usage of x87 easier, all *MMX* registers are temporaries, so the FEMMS instruction may be used.

`MM0`, `MM1` Temporary registers used to return first two doublewords of the return value.

`MM2` – `MM8` Temporary registers.

`st(0)` – `st(8)` Temporary registers not available across function calls

Special purpose registers

All special purpose registers should be handled as global registers with the exception of the direction flag in EFLAGS register. The value of the direction flag must be clear at the entry of a function. (*Editor:* Do we want to define values of fpu and SSE control words?)

3.2.2 The Stack Frame

In addition to registers, each function has a frame on the run-time stack. This stack grows downwards from high addresses, figure 3.3 shows the stack organization.

Functions may expect the end of the input argument area to be aligned to a 16 byte boundary — this means that value $(\mathtt{RSP} - 8)$ is always a multiple of 16 at the entry point. This alignment must be preserved by all called functions that

Figure 3.3: Stack Frame

Frame	Contents	Size	Address
	incoming arguments: doubleword n	8 bytes	RBP $+ 16 + 8n$
	...		
Previous	incoming arguments: doubleword 0	8 bytes	RBP $+ 16$
Current	return address	8 bytes	RBP $+ 8$
	previous RBP value	8 bytes	RBP
	local save area	unspecified	RBP $- 8$
	...		
	outgoing arguments: doubleword 0	8 bytes	
	...		
	outgoing arguments: doubleword n	8 bytes	RSP

use the standard calling convention. The frames of functions with local calling conventions may or may not obey this rule so algorithms for stack unwinding cannot rely on it.

At any point of program execution, the stack pointer, RSP, shall point to the end of the latest allocated stack frame.

The organization of the *local save area* is implementation specific. Usual practice is to save clobbered global registers in the prologue at the beginning or end of the save area and use the rest for saving local object and spilled variables. Saving registers at the beginning is harder to implement, since the offset from RBP is not known until after the register allocation, but overall results in shorter and faster prologues and epilogues allowing to allocate and deallocate the outgoing arguments area by the same instruction used to (de)allocate the *local save area*.

The usage of RBP as a frame pointer may be avoided by using RSP instead. This saves two instructions in the prologue and epilogue and gives another global general purpose register. See details about stack traceback algorithm for conditions when this optimization is allowed. (*Editor:* The stack traceback algorithm has not been specified yet and we are not sure if we need one at all.)

3.2.3 Classes

The x86-64 CPU has three register classes available for parameter and return value passing — the integer (general purpose) registers, SSE registers and *MMX* registers. All integral types and pointer belong to the class *Int*, all floating point values (including the 128bit type) and type `__m128` belongs to the class *SSE* and finally the type `__mm64` to class *MMX*.

Aggregate and union types are split to doublewords and each doubleword has assigned its own class. To assign the class of doubleword *X* the first matching rule of the following is used.

1. (Aggregates only) In case the doubleword *X* contains upper half field of `__m128` type, the class is *SSE*.

2. In case all fields of an aggregate (or all subtypes of a union) crossing the word *X* have the same class *Y*, the class is *Y*.

3. In case all fields of an aggregate (or all subtypes of a union) crossing the word *X* have the same class *SSE* or *SSE2*, the class is *SSE*.

4. If nothing matches, use class is *Int*.

Doubleword *X* of an aggregate gets assigned class *Y* if and only if all the fields crossing doubleword *X* belongs to same class *Y*. If no field crosses the given doubleword, its type is *None*. When no class fits the rules above, class *Int* is used. Exception is also (*Editor:* Jan, something is missing here — please fill in the rest of the sentence.)

Similarly doubleword *X* of a union gets assigned class *Y* if and only if each type in the union either does not cross *X* or its class for doubleword *X* is *Y*.

3.2.4 Parameter Passing Conventions

It is generally more efficient to pass arguments to called functions in registers than to push them onto the stack. Even with good support for memory operands and `push` instruction, passing arguments in registers reduces memory bandwidth as well as simplifies the cleanup after function call. The number of registers implemented in the processor architecture naturally limits the number of arguments passed in this manner.

For x86-64 up to six general purpose registers and six SSE registers are used for parameter passing. If fewer (or no) arguments are passed, the unneeded registers will contain undefined value at the entry of function. (*Editor:* The number of registers used for passing is subject to change.)

MMX registers are not used to pass `__mm64` type, since *MMX* usage is depreached and doing so simplifies functions with variable number of arguments. Instead all arguments of type `__mm64` as well as arguments that do not fit into the available registers are passed on the runtime stack.

The aggregates and unions shorter than 16 bytes are passed directly using registers, while others by reference. The caller is expected to make a copy of the object so the callee is allowed to freely modify the contents. The caller may not copy the object in case it is known to be unused after a function call, it does have proper alignment and no aliases. The aggregate or union, when passed in registers is represented equally as if it were passed at the stack slot.

The following algorithm is used to determine where data is passed for C language. For this purpose consider arguments as ordered from left (first argument) to right (last arguments), although the order of evaluation is unspecified. In this algorithm `sr` contains the first available SSE register and `gr` contains the next available general purpose register and `ap` is used as a pointer to the "output argument area" on the stack. As written, the algorithm is not directly applicable, since it fills the stack in the opposite direction. In reality a two pass algorithm is required. The first pass computes positions for output arguments and the second pass actually pushes the arguments onto the stack and/or loads them into registers.

Proposal 1: "Smart" mapping of structures

Initialize: Set `sr` $= 1$, `gr` $= 1$, `ap` $= 0$

Scan: If there are no more arguments, go to "terminate". Otherwise select one of the following rules depending on the type of the next argument:

Simple integer argument: A simple integer argument is one of the following:

- One of the simple integer types, maximal 64 bits long.
- A pointer.
- Aggregates and unions greater than 16 bytes should be treated as pointer to object, or pointer to a copy of the objects if necessary to enforce call-by-value semantics.

- Aggregates and unions maximal 8 bytes long with class *Int*.

If `gr` $>= 6$ go to other, otherwise pick the `gr`-th register from the sequence: `RAX`, `RDX`, `RCX`, `RBX`, `RSI`, `RDI` and assign it to the current argument. Set `gr` to `gr` $+ 1$.

Simple SSE argument: A simple SSE argument is one of the following:

- One of the simple floating point types.
- An argument of type `__m128`.
- Aggregates and unions maximal 8 bytes long with class *SSE*.
- Aggregates and unions greater than 8 bytes and smaller than 17 bytes with class *SSE* of the first doubleword and class *SSE* of the second doubleword.

If `sr` $>= 6$ go to other, otherwise pick the `sr`-th register from the sequence: `XMM0`, `XMM1`, `XMM2`, `XMM3`, `XMM4`, `XMM5`, `XMM6` and assign it to the current argument. Set `sr` to `sr` $+ 1$.

Double integer argument A double integer argument is one of the following:

- An argument of type `__int128`.
- Aggregates and unions greater than 8 bytes and smaller than 17 bytes with class *Int* for both doublewords.

If `gr` $>= 5$ go to other, otherwise pick the `gr`-th register from the sequence: `RAX`, `RDX`, `RCX`, `RBX`, `RSI`, `RDI` and assign it together with the next register to the current argument. Set `gr` to `gr` $+ 2$.

Double SSE argument A double SSE argument is one of the following:

- Aggregates and unions greater than 8 bytes and smaller than 17 bytes with class *SSE* for both doublewords.

If `sr` $>= 5$ go to other, otherwise pick the `sr`-th register from the sequence: `XMM0`, `XMM1`, `XMM2`, `XMM3`, `XMM4`, `XMM5`, `XMM6` and assign it together with the next register to the current argument. Set `sr` to `sr` $+ 2$.

Integer-SSE type argument An Integer-SSE type argument is one of the following:

- Aggregates and unions greater than 8 bytes and smaller than 17 bytes with class *Int* for the first doubleword and *SSE* for the second doubleword.

If `sr` $>= 6$ or `gr` $>= 6$ go to other, otherwise pick the `gr`-th register from the sequence: RAX, RDX, RCX, RBX, RSI, RDI and assign it the first doubleword of the current argument and pick the `sr`-th register from the sequence: XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6 and assign it the second doubleword of the current argument. Set `gr` to `gr` $+ 1$ and set `sr` to `sr` $+ 1$.

SSE-Integer type argument An SSE-Integer type argument is one of the following:

- Aggregates and unions greater than 8 bytes and smaller than 17 bytes with class *SSe* for the first doubleword and *Int* for the second doubleword.

If `sr` $>= 6$ or `gr` $>= 6$ go to other, otherwise pick the `sr`-th register from the sequence: XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6 and assign it the first doubleword of the current argument and pick the `gr`-th register from the sequence: RAX, RDX, RCX, RBX, RSI, RDI and assign it the second doubleword of the current argument. Set `gr` to `gr` $+ 1$ and set `sr` to `sr` $+ 1$.

Go to “Scan”.

Other: Arguments not handled otherwise are passed at stack frame. RSP is the pointer to the end of the outgoing argument area and we do expect RSP to be aligned to a 16 byte boundary and the outgoing area large enough to fit all the arguments (this means the value `ap` bytes for the final value of `ap` once algorithm terminates). To compute the address of a given argument, round `ap` to the nearest multiple of the operand’s alignment in bytes (minimally to 8 bytes) and assign location `ap` $+$ RSP to the argument and increase `ap` by the size of the argument in bytes. Go to “Scan”.

Terminate: Round `ap` up to a multiple of 16. Terminate the algorithm.

As an example assume the function call as shown below:

```
int a,b,c,d,e;
float A,B;
double C,D;
long double E;
__mm64 mm;
```

Table 3.1: Register Allocation For Parameter Passing.

integer register	contents	SSE register	contents	stack slot	contents
RAX	small.a	XMM0	small.b	RSP	mm
RDX	&large	XMM1	A	RSP $+ 8$	e
RCX	a	XMM2	B		
RBX	b	XMM3	C		
RSI	c	XMM4	D		
RDI	d	XMM5	E		

```
struct small {long a; double b;} small;
struct large {int a,b,c,d,e,f;} small;
func (small, large, mm, a, b, c, d, e, A, B, C, D, E);
```

The results of register allocation for parameter passing is shown in table 3.1.

Proposal 2: Structures always mapped to integers

Initialize: Set `sr` $= 1$, `gr` $= 1$, `ap` $= 0$

Scan: If there are no more arguments, go to "terminate". Otherwise select one of the following rules depending on the type of the next argument:

Simple integer argument: A simple integer argument is one of the following:

- One of the simple integer types, maximal 64 bits long.
- A pointer.
- Aggregates and unions greater than 16 bytes should be treated as pointer to object, or pointer to a copy of the objects if necessary to enforce call-by-value semantics.
- Aggregates and unions maximal 8 bytes long with class *Int*.

If `gr` $>= 6$ go to other, otherwise pick the `gr`-th register from the sequence: `RAX`, `RDX`, `RCX`, `RBX`, `RSI`, `RDI` and assign it to the current argument. Set `gr` to `gr` $+ 1$.

Simple SSE argument: A simple SSE argument is one of the following:

- One of the simple floating point types.

- An argument of type `__m128`.
- Aggregates and unions maximal 8 bytes long with class *SSE*.
- Aggregates and unions greater than 8 bytes and smaller than 17 bytes with class *SSE* for the first doubleword and class *SSE2* for the second doubleword.

If `sr` $>= 6$ go to other, otherwise pick the `sr`-th register from the sequence: `XMM0`, `XMM1`, `XMM2`, `XMM3`, `XMM4`, `XMM5`, `XMM6` and assign it to the current argument. Set `sr` to `sr` $+ 1$.

Double integer argument A double integer argument is one of the following:

- An argument of type `__int128`.
- Aggregates and unions greater than 8 bytes and smaller than 17 bytes.

If `gr` $>= 5$ go to other, otherwise pick the `gr`-th register from the sequence: `RAX`, `RDX`, `RCX`, `RBX`, `RSI`, `RDI` and assign it together with the next register to the current argument. Set `gr` to `gr` $+ 2$.

Go to “Scan”.

Other: Arguments not handled otherwise are passed at stack frame. `RSP` is the pointer to the end of outgoing argument area and we do expect `RSP` to be aligned to a 16 byte boundary and the outgoing area space large enough to fit all the arguments (this means the value `ap` bytes for the final value of `ap` once algorithm terminates). To compute the address of a given argument, round `ap` to the nearest multiple of the operand’s alignment in bytes (minimal to 8 bytes) and assign location `ap` $+$ `RSP` to the argument and increase `ap` by the size of the argument in bytes. Go to “Scan”.

Terminate: Round `ap` up to a multiple of 16. Terminate the algorithm.

As an example assume the function call as shown below:

```
int a,b,c,d,e;
float A,B;
double C,D;
long double E;
__mm64 mm;
```

Table 3.2: Register Allocation For Parameter Passing.

integer register	contents	SSE register	contents	stack slot	contents
RAX	small.a	XMM0	A	RSP	mm
RDX	small.b	XMM1	B	RSP $+ 8$	e
RCX	&large	XMM2	C	RSP $+ 16$	d
RBX	a	XMM3	D		
RSI	b	XMM4	E		
RDI	c	XMM5	undefined		

```
struct small {long a; double b;} small;
struct large {int a,b,c,d,e,f;} small;
func (small, large, mm, a, b, c, d, e, A, B, C, D, E);
```

The results of register allocation for parameter passing is shown in table 3.2.

3.2.5 Variable Argument Lists

Some otherwise portable C programs depend on the argument passing scheme, implicitly assuming that 1) all arguments are passed on the stack, and 2) arguments appear in increasing order on the stack. Programs that make these assumptions never have been portable, but they have worked on many implementations. However, they do not work on the x86-64 architecture because some arguments are passed in registers. Portable C programs use the header files `stdarg.h` or `varargs.h` to deal with variable argument lists on x86-64 and other machines as well.

(*Editor:* Use flag to signalize SSE registers?)

3.2.6 Return Values

Proposal 1: "Smart" mapping of structures

Scalars, complex numbers, aggregates and unions not exceeding 16 bytes or no value are returned directly. First one or two classes are assigned to a given value and then the relevant registers are used as discussed in subsection 3.2.1 leaving the rest of registers undefined.

Scalar values are returned in a single register. One of the registers `RAX`, `XMM0` and `MM0` is chosen according to its class. `__int128` is returned in `RDX`:`RAX`.

The first doubleword of unions, aggregates and real part of complex values not exceeding 8 bytes is returned the same way as simple scalar registers. The second doubleword (if available) is returned in one of the following registers: `RAX`, `XMM0` and `MM1` depending on their class. Class *SSE* is returned in the upper half of register `XMM0`.

Complex numbers, aggregates and unions exceeding 16 bytes are stored in an area prepared by the caller and passed to the function as "hidden" first parameter. The contents of all return registers is left undefined. (*Editor:* i386 abi returns the pointer to the area to simplify caller. Is that a good idea?)

Proposal 2: Integer-only mapping of structures

The `__mm64` type is returned using `MM0`. Complex numbers, aggregates and unions exceeding 16 bytes are stored in an area prepared by the caller and passed to the function as "hidden" first parameter. The contents of all return registers is left undefined.

Other values are returned as if they were passed as first argument to the function.

3.3 Operating System Interface

3.3.1 Virtual Address Space

Although the x86-64 architecture uses 64-bit pointers, implementations are only required to handle 48-bit addresses. Therefore, conforming processes may only use addresses from `0x0000000000000000` to `0x0000ffffffffffff`.

No other changes required.

3.3.2 Page Size

Systems are permitted to use any page size between 4KB and 64KB, inclusive.

No other changes required.

3.3.3 Virtual Address Assignments

Conceptually processes have the full address space available. In practice, however, several factors limit the size of a process.

- The system reserves a configuration dependent amount of virtual space.
- The system reserves a configuration dependent amount of space per process.
- A process whose size exceeds the system's available combined physical memory and secondary storage cannot run. Although some physical memory must be present to run any process, the system can execute processes that are bigger than physical memory, paging them to and from secondary storage. Nonetheless, both physical memory and secondary storage are shared resources. System load, which can vary from one program execution to the next, affects the available amount.

Figure 3.4: Virtual Address Configuration

`0xffffffffffffffff`	Reserved system area ...	End of memory
`0x80000000000`	... Dynamic segments	
`0`	... Process segments	Beginning of memory

Although applications may control their memory assignments, the typical arrangement appears in figure 3.5.

Figure 3.5: Conventional Segment Arrangements

`0x80000000000`	... Dynamic segments
	Stack segment ...
	... Data segments
`0x10000`	... Text segments
`0`	Unmapped

3.4 Coding Examples

The following sections show only the difference to the i386 ABI.

3.4.1 Position-Independend Function Prologue

x86-64 does not need any function prologue for calculating the global offset table address since it does not have an explicit GOT pointer.

3.4.2 Data Objects

Not done yet.

3.4.3 Function Calls

Figure 3.6: Position-Independent Direct Function Call

C	Assembly
extern void function (); function ();	.globl function call function@PLT

Figure 3.7: Position-Independent Indirect Function Call

C	Assembly

extern void (*ptr) ();
extern void name ();
ptr = name;
	.globl ptr, name
	
	movl ptr@GOTPCREL(%rip), %rax
	movl name@GOTPCREL(%rip), %rdx
	movl %rdx, (%rax)

(*ptr)();	movl ptr@GOTPCREL(%rip), %rax call *(%rax)

Chapter 4

Object Files

4.1 ELF Header

4.1.1 Machine Information

For file identification in e_ident, the x86-64 architecture requires the following values.

Table 4.1: x86-64 Identification

Position	Value
e_ident[EI_CLASS]	ELFCLASS64
e_ident[EI_DATA]	ELFDATA2LSB

Processor identification resides in the ELF header's e_machine member and must have the value EM_X8664.[1]

4.2 Sections

No changes required.

[1] The value of this identifier is 62.

4.3 Symbol Table

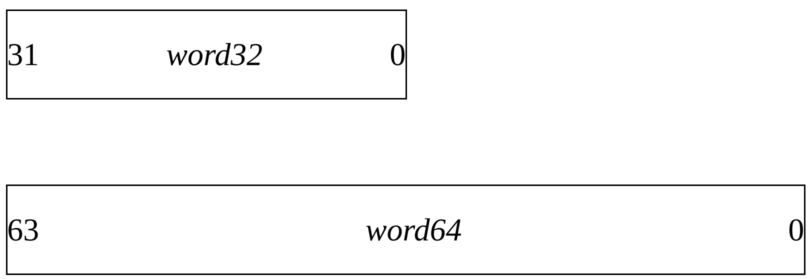
No changes required.

4.4 Relocation

4.4.1 Relocation Types

The x86-64 ABI adds one additional field:

Figure 4.1: Relocatable Fields



word32 This specifies a 32-bit field occupying 4 bytes with arbitrary byte alignment. These values use the same byte order as other word values in the x86-64 architecture.

word64 This specifies a 64-bit field occupying 8 bytes with arbitrary byte alignment. These values use the same byte order as other word values in the x86-64 architecture.

The x86-64 ABI architectures uses only `Elf64_Rel` relocation entries.

Figure 4.2: Relocation Types

Name	Value	Field	Calculation
R_X8664_NONE	0	none	none
R_X8664_64	1	*word64*	S + A
R_X8664_PC32	2	*word32*	S + A - P
R_X8664_GOT32	3	*word32*	G + A
R_X8664_PLT64	4	*word64*	L + A - P
R_X8664_COPY	5	none	none
R_X8664_GLOB_DAT	6	*word64*	S
R_X8664_JMP_SLOT	7	*word64*	S
R_X8664_RELATIVE	8	*word64*	B + A
R_X8664_GOTPCREL	9	*word64*	G + GOT + A - P

The special semantics for these relocation types are identical to those used for the Intel386 architecture. [2] [3]

The R_X8664_GOTPCREL relocation has different semantics from the i386 R_I386_GOTPC relocation. In particular, because the x86-64 architecture has an addressing mode relative to the instruction pointer, it is possible to load an address from the GOT using a single instruction. The calculation done by the R_X8664_GOTPCREL relocation gives the difference between the location in the GOT where the symbol's address is given and the location where the relocation is applied.

[2]Even though the x86-64 architecture supports IP-relative addressing modes, a GOT is still required since the address from a particular instruction to a particular data item cannot be known by the static linker.

[3]Note that the x86-64 architecture assumes that offsets into GOT are 32-bit values, not 64-bit values. This choice means that a maximum of $2^{32}/8 = 2^{29}$ entries can be placed in the GOT. However, that should be more than enough for most programs. In the event that it is not enough, the linker could create multiple GOTs. Because 32-bit offsets are used, loads of global data do not require loading the offset into a displacement register; the base plus immediate displacement addressing form can be used.

Chapter 5

Program Loading and Dynamic Linking

5.1 Program Loading

No changes required.

5.2 Dynamic Linking

Dynamic Section

No changes required.

Global Offset Table

The only difference is in the size of the global offset table, it contains 64 bit addresses.

Figure 5.1: Global Offset Table

```
extern Elf64_Addr _GLOBAL_OFFSET_TABLE_ [];
```

Function Addresses

No changes required.

Procedure Linkage Table

(*Editor:* This is ia32 enhanced to 64 bits without an explicit GOT register.)

Much as the global offset table redirects position-independent address calculations to absolute locations, the procedure linkage table redirects position-independent function calls to absolute locations. The link editor cannot resolve execution transfers (such as function calls) from one executable or shared object to another. Consequently, the link editor arranges to have the program transfer control to entries in the procedure linkage table. On the x86-64 architecture, procedure linkage tables reside in shared text, but they use address in the private global offset table. The dynamic linker determines the destinations' absolute addresses and modifies the global offset table's memory image accordingly. The dynamic linker thus can redirect the entries without compromising the position-independence and shareability of the program's text. Executable files and shared object files have separate procedure linkage tables.

Figure 5.2: Absolute Procedure Linkage Table

```
.PLT0:   pushl  got_plus_8; GOT[1]
         jmp    got_plus_16 ; GOT[2]
         nop
         nop
         nop
         nop
.PLT1:   jmp    *name1_in_GOT
.PLT1a   pushl  $offset
         jmp    .PLT0@PC
.PLT2:   jmp    *name2_in_GOT
         pushl  $offset
         jmp    .PLT0@PC
         ...
```

Figure 5.3: Position-Independent Procedure Linkage Table

```
.PLT0:  pushl  GOT+8(%rip); GOT[1]
        jmp    GOT+16(%rip) ; GOT[2]
        nop
        nop
        nop
        nop
.PLT1:  jmp    *name1@GOTPC(%rip)
        pushl  $offset
        jmp    .PLT0@PC
.PLT2:  jmp    *name2@GOTPC(%rip)
        pushl  $offset
        jmp    .PLT0@PC
        ...
```

Following the steps below, the dynamic linker and the program "cooperate" to resolve symbolic references through the procedure linkage table and the global offset table.

1. When first creating the memory image of the program, the dynamic linker sets the second and the third entries in the global offset table to special values. Steps below explain more about these values.

2. Each shared object file in the process image has its own procedure linkage table, and control transfers to a procedure linkage table entry only from within the same object file. (*Editor:* Should we remove this completely?)

3. For illustration, assume the program calls `name1`, which transfers control to the label `.PLT1`.

4. The first instruction jumps to the address in the global offset table entry for `name1`. Initially the global offset table holds the address of the following `pushl` instruction, not the real address of `name1`.

5. After pushing the relocation offset, the program then jumps to `.PLT0`, the first entry in the procedure linkage table. The `pushl` instruction places the

value of the second global offset table entry (GOT+8) on the stack, thus giving the dynamic linker one word of identifying information. The program then jumps to the address in the third global offset table entry (GOT+16), which transfers control to the dynamic linker.

6. When the dynamic linker receives control, it unwinds the stack, looks at the designated relocation entry, finds the symbol's value, stores the "real" address for `name1` in its global offset table entry, and transfers control to the desired destination.

7. Subsequent executions of the procedure linkage table entry will transfer directly to `name1`, without calling the dynamic linker a second time. That is, the `jmp` instruction at `.PLT1` will transfer to `name1`, instead of "falling through" to the `pushl` instruction.

The `LD_BIND_NOW` environment variable can change the dynamic linking behavior. If its value is non-null, the dynamic linker evaluates procedure linkage table entries before transferring control to the program. That is, the dynamic linker processes relocation entries of type `R_X8664_JMP_SLOT` during process initialization. Otherwise, the dynamic linker evaluates procedure linkage table entries lazily, delaying symbol resolution and relocation until the first execution of a table entry.

5.2.1 Program Interpreter

There is one valid program interpreter for programs conforming to the x86-64 ABI:

```
/usr/lib/ld64.so.1
```

5.2.2 Initialization and Termination Functions

The implementation is responsible for executing the initialization functions specified by `DT_INIT`, `DT_INIT_ARRAY` , and `DT_PREINIT_ARRAY` entries in the executable file and shared object files for a process, and the termination (or finalization) functions specified by `DT_FINI` and `DT_FINI_ARRAY`, as specified by the *System V ABI*. The user program plays no further part in executing the initialization and termination functions specified by these dynamic tags.

Chapter 6

Libraries

Not done yet.

Chapter 7

Development Environment

No changes required.

Chapter 8

Execution Environment

Not done yet.

Chapter 9

Conventions

(*Editor:* This chapter is used to document some features special to the x86-64 ABI. The different sections might be moved to another place or removed completely.)

9.1 GOT pointer and IP relative addressing

A basic difference between the i386 ABI and the x86-64 ABI is the way the GOT table is found. The i386 ABI, like (most) other processor specific ABIs, uses a dedicated register (`EBX`) to address the base of the GOT table. The function prologue of every function needs to set up this register to the correct value. The x86-64 processor family introduces a new IP-relative addressing mode which is used in this ABI instead of a using a dedicated register.

On x86-64 the GOT table contains 64 bit entries.

9.2 Execution of 32bit programs

The x86-64 is able to execute 64 bit x86-64 and also 32 bit ia32 programs. Libraries and programs conforming to the x86-64 ABI will live in the normal places like `/lib`, `/usr/lib` and `/usr/bin`. Programs following the 32 bit ia32 ABI (known as i386 ABI), will use `ia32` subdirectories for the libraries, e.g `/lib/ia32` and `/usr/lib/ia32`. Binaries will remain in the same location. The location of the program interpreter for ia32 binaries is not changed.

9.3 C++

For the C++ ABI we will use the ia64 C++ ABI and instantiate it appropriatly. The current draft of that ABI is available at:
`http://reality.sgi.com/dehnert_engr/cxx/abi-eh.html`